Building Your First "Deep" Neural Network Introduction to Backpropagation Who invented backpropagation? — JURGEN SCHMIDHUBER 99 IN THIS CHAPTER Licensed to Asif Qamar 100 Chapter 6 I Building Your First "Deep" Neural Network The Street Light Problem A toy problem for us to consider how a network learns entire datasets. Consider yourself approaching a street corner in a foreign country. As you approach, you look up and realize that the street light is quite unfamiliar. How can you know when it is safe to cross the street? You can know when it is safe to cross the street by interpreting the streetlight. How ever, in this case, we don't know how to interpret it! Which "light" combinations indicate when it is time to walk? Which indicate when it is time to stop? To solve this problem, you might sit at the street corner for a few minutes observing correlation between each light combination and whether or not people around you choose to walk or stop. You take a seat and record the following pattern. Ok, nobody walked at the first light. At this point you're thinking, "man, this pattern could be anything. The left light or the right light could be correlated with stopping, or the central light could be correlated with walking." There's no way to know. Let's take another datapoint. People walked! Ok, so something changed with this light that changed the signal. The only thing we know for sure is that the far right light doesn't seem to indicate one way or another. Perhaps it is irrelevant. Let's collect another datapoint. STOP WALK Licensed to Asif Qamar The Street Light Problem 101 STOP Now we're getting somewhere! Only the middle light changed this time, and we got the opposite pattern. Our working hypothesys is that the middle light indicates when people feel safe to walk. Over the next few minutes, we recorded the following six light patterns, noting when people seemed to walk or stop. Do you notice a pattern overall? As hypothesized on the previous page, there is a perfect correlation between the middle (criss-cross) light and whether or not it is safe to walk. You were able to learn this pattern by observing all of the individual datapoints and searching for correlation. This is what we're going to train our neural network to do. STOP WALK WALK WALK STOP STOP Licensed to Asif Qamar 102 Chapter 6 I Building Your First "Deep" Neural Network Preparing our Data Neural Networks Don't Read Streetlight In the previous chapters, we learned about supervised algo rithms. We learned that they can take one dataset and turn it into another. More importantly, they can take a dataset of what we know and turn it into a dataset of what we want to know. So, how do we train a supervised neural network? Well, we present it with two datasets and ask it to learn how to transform one into the other. Think back to our streetlight problem. Can you identify two datasets? Which one do we always know? Which one do we want to know? We do indeed have two datasets. On the one hand, we have six streetlight states. On the other hand, we have 6 observations of whether people walked or not. These are our two datasets. So, we can train our neural network to convert from the data set we know to the dataset that we want to know. In this particular "real world example", we know the state of the streetlight at any given time, and we want to know whether it is safe to cross the street. So, in order to prepare this data for our neural network, we need to first split it into these two groups (what we know and what we want to know). Note that we could attempt to go backwards if we swapped which dataset was in which group. For some problems, this works. STOP WALK WALK WALK STOP STOP What We Know What We Want to Know Licensed to Asif Qamar Matrices and the Matrix Relationship 103 Matrices and the Matrix Relationship Translating your streetlight into math. Math doesn't understand streetlights. As mentioned in the pre vious section, we want to teach our neural network to translate a street light pattern into the correct stop/walk pattern. The operative word here is pattern. What we really want to do is mimick the pattern of our streetlight in the form of numbers. Let me show you what I mean. Streetlights 1 0 1 0 1 1 0 0 1 1 1 1 0 1 1 1 0 1 Streetlight Pattern Notice in the matrix on the right that we have mimicked the pattern from our streetlights in the form of 1s and 0s. Notice that each of the lights gets a column (3 columns total since there are three lights). Notice also that there are 6 rows representing the 6 different streetlights that we observed. This structure of 1s and 0s is called a matrix. Furthermore, this relationship between the

rows and columns is very common in matri ces, especially matrices of data (like our streetlights). In data matrices, it is convention to give each recorded example a single row. It is also convention to give each thing being recorded a single column. This makes it easy to read. So, a column contains every state we recorded a thing in. In this case, a column contains every on/off state we recorded of a partic ular light. Each row contains the simultaneous state of every light at a particular moment in time. Again, this is common. Licensed to Asif Qamar 104 Chapter 6 I Building Your First "Deep" Neural Network Good data matrices perfectly mimick the outside world. Our data matrix doesn't have to be all 1s and 0s. What if the streetlights were on "dimmers" and they turned on and off at varying degrees of intensity. Perhaps our streetlight matrix would look more like this. Matrix A above is a perfectly valid matrix. It is mimicking the patterns that exist in the real world (streetlight) so that we can ask our computer to interpret them. Would the following matrix still be valid? Streetlights .9 .0 1 .2 .8 1 .1 .0 1 .8 .9 1 .1 .7 1 .9 .1 0 Streetlight Matrix A Streetlights 9 0 10 2 8 10 1 0 10 8 9 10 1 7 10 9 1 0 Streetlight Matrix B In fact, this matrix (B) is still valid. It adquality captures the relationships between various training examples (rows) and lights (columns). Note that "Matrix A" * 10 == "Matrix B" (A * 10 == B). Th is actually means that these matrices are scalar multiples of each other. Licensed to Asif Qamar Preparing our Data 105 Matrix A and B both contain the same underlying pattern. Th e important takeaway here is that there are an infi nite number of matrices that perfectly refl ect the streetlight patterns in our dataset. Even the one below is still perfect. Streetlights 18 0 20 4 16 20 2 0 20 16 18 20 2 14 20 18 2 0 Streetlight Matrix C It's important to recognize that "the underlying pattern" is not the same as "the matrix". It's a property of the matrix. In fact, it's a property of all three of these matrices (A, B, and C). Th e pattern is what each of these matrices is expressing. Th e pattern also existed in the streetlights. Th is input data pattern is what we want our neural network to learn to transform into the output data pattern. However, in order to learn the output data pattern, we also need to capture the pat tern in the form of a matrix. Let's do that below. STOP WALK WALK WALK STOP STOP 0 1 0 1 1 0 Note that we could reverse the 1s and 0s here and the output matrix would still be capturing the underlying STOP/WALK pattern that's present in our data. We know this because regardless of whether we assign a 1 to WALK or to STOP, we can still decode the 1s and 0s into the underlying STOP/WALK pattern. We call this resulting matrix a lossless representation because we can can perfectly convert back and forth be tween our stop/walk notes and the matrix. Licensed to Asif Qamar 106 Chapter 6 I Building Your First "Deep" Neural Network Creating a Matrix or Two in Python Importing our matrices into Python So, we've converted our streetlight pattern into a matrix (the one with just 1s and 0s). Now, we want to create that matrix (and more importantly, it's underlying pattern) in Python so that our neural network can read it. Python has a special library built just for handling matrices called numpy. Let's see it in action. import numpy as np streetlights = np.array([1, 0, 1], [0, 1, 1], [0, 0, 1], [1, 1, 1], [0, 1, 1], [1, 0, 1]]) If you're a regular Python user, something should be very striking from this code. A matrix is really just a list of lists! It's an array of arrays! What is numpy? Numpy is really just a fancy wrapper for an array of arrays that gives us special matrix-oriented functions. Let's create a numpy matrix for our output data too. walk _ vs _ stop = np.array([0], [1], [0], [1], [1], [0]]) So, what will we want our neural network to do? Well, we will want it to take our streetlights matrix and learn to transform it into our walk _ vs _ stop matrix. More importantly, we will want our neural network to take any matrix containing the same underlying pattern as streetlights and transform it into a matrix that contains the un derlying pattern of walk _ vs _ stop. More on that later. Let's start by trying to transform streetlights into walk _ vs _ stop using a neural network. streetlights walk _ vs _ stop Neural Network Licensed to Asif Qamar Building Our Neural Network 107 Building Our Neural Network Ok, so we've been learning about neural networks for several chapters now. We've got a new dataset, and we're going to create a neural network to solve it. Below, I've written out some example code to learn the first streetlight pattern. This should look very familiar. import numpy as

np weights = np.array([0.5,0.48,-0.7]) alpha = 0.1 streetlights = np.array([[1, 0, 1], [0, 1, 1], [0, 0, 1], [1, 1, 1], [0, 1, 1], [1, 0, 1]]) walk_vs_stop = np.array([0, 1, 0, 1, 1, 0]) input = streetlights[0] # [1,0,1] goal_prediction = walk_vs_stop[0] # equals 0... i.e. "stop" for iteration in range(20): prediction = input.dot(weights) error = (goal_prediction - prediction) ** 2 delta = prediction - goal_prediction weights = weights - (alpha * (input * delta)) print "Error:" + str(error) + " Prediction:" + str(prediction) Perhaps this code example will bring back several nuances we learned in Chapter 3. First, the use of the function "dot" was a way to perform a dot product (weighted sum) between two vectors. However, not included in Chapter 3 is the way that numpy matrices can perform elementwise addition and multiplication. import numpy as np a = np.array([0,1,2,1]) b = np.array([2,2,2,3]) print a*b #elementwise multiplication print a+b #elementwise addition print a * 0.5 # vector-scalar multiplication print a + 0.5 # vector-scalar addition One could say that numpy makes these operations very easy. When you put a "+" sign between two vectors, it does what you would expect it to. It adds the two vectors together. Other than these nice numpy operators and our new dataset, the neural network above is the same as ones we built before. Licensed to Asif Qamar 108 Chapter 6 I Building Your First "Deep" Neural Network Learning the whole dataset! So... in the last few pages... we've only been learning one streetlight. Don't we want to learn them all? So far in this book, we've trained neural networks that learned how to model a sin gle training example (input -> goal_pred pair). However, now we're trying to build a neural network that tells us "whether or not it is safe to cross the street". We need it to know more than one streetlight! How do we do this? Well... we train it on all the streetlights at once! import numpy as np weights = np.array([0.5,0.48,-0.7]) alpha = 0.1 streetlights = np.array([[1, 0, 1], [0, 1, 1], [0, 0, 1], [1, 1, 1], [0, 1, 1], [1, 0, 1]]) walk_vs_stop = np.array([0, 1, 0, 1, 1, 0]) input = streetlights[0] # [1,0,1] goal_prediction = walk_vs_stop[0] # equals 0... i.e. "stop" for iteration in range(40): error_for_all_lights = 0 for row_index in range(len(walk_vs_stop)): input = streetlights[row_index] goal_prediction = walk_vs_stop[row_index] prediction = input.dot(weights) error = (goal_prediction - prediction) ** 2 error_for_all_lights += error delta = prediction - goal_prediction weights = weights - (alpha * (input * delta)) print "Prediction:" + str(prediction) print "Error:" + str(error_for_all_lights) + "\n" Error:2.6561231104 Error:0.962870177672 ... Error:0.000614343567483 Error:0.000533736773285 Licensed to Asif Qamar Full / Batch / Stochastic Gradient Descent 109 Full / Batch / Stochastic Gradient Descent Stochastic Gradient Descent - Updating weights one example at a time. As it turns out, this idea of learning "one example at a time" is a variant on Gradient Descent called Stochastic Gradient Descent, and it is one of the handful of methods that can be used for learning an entire dataset. How does Stochastic Gradient Descent work? As exemplified on the previous page, it simply performs a prediction and weight update for each training example separately. In other words, it takes the first streetlight, tries to predict it, calculates teh weight_delta, adn updates the weights. Then it moves onto the second streetlight, etc. It iterates through the entire dataset many times until it can find a weight configuration that works well for all of the training examples. (Full) Gradient Descent - Updating weights one dataset at a time. As it turns out, another method for learning an entire dataset is just called Gradient De scent (or "Average/Full Gradient Descent" if you like). Instead of updating the weights once for each training example, the network simply calculates the average weight_delta over the entire dataset, only actually changing the weights each time it computes a full average. Batch Gradient Descent - Updating weights after "n" examples. This will be covered in more detail later, but there is also a third configuration that sortof "splits the difference" between Stochastic Gradient Descent and Full Gradient Descent. Instead of updating the weights after just one or after the entire dataset of examples, you choose a "batch size" (typically between 8 and 256) after which the weights are updated. We will discuss this more later in the book, but for now, simply recognize that on the pre vious page we created a neural network that can learn our entire "Streetlights" dataset by

training on each example one at a time. Licensed to Asif Qamar 110 Chapter 6 I Building Your First "Deep" Neural Network Neural Networks Learn Correlation What did our last neural network learn? We just got done training a single-layer neural network to take a streetlight pattern and identify whether or not it was safe to cross the street. Let's take on the neural network's per spective for a moment. Th e neural network doesn't know that it was processing streetlight data. All it was trying to do was identify which input (out of the 3 possible) correlated with the output. You can see that it correctly identifi ed the middle light by analyzing the fi nal weight positions of the network. walk/stop .01 1.0 input -.0 output Notice that the middle weight is very near 1 while the far left and right weights are very near 0. At a high level, all the iterative, complex processes for learning we identifi ed actually accomplished something rather simple. Th e network identifi ed correlation between the middle input and output. Th e correlation is located wherever the weights were set to high numbers. Inversely, randomness with respect to the output was found at the far left and far right weights (where the weight values are very near 0). How did it identify correlation? Well, in the process of Gradient Descent, each training example either asserts up pressure or down pressure on our weights. On average, there was more up pressure for our middle weight and more down pressure for our other weights. Where does the pressure come from? Why is it diff erent for diff erent weights? Licensed to Asif Qamar Up and Down Pressure 111 Up and Down Pressure It comes from our data. Each neuron is individually trying to correctly predict the output given the input. For the most part, each neuron ignores all the other neurons when attempting to do so. The only cross communication occurs in that all three weights must share the same error mea sure. Our weight update is nothing more than taking this shared error measure and multiply ing it by each respective input. Why do we do this? Well, a key part of why neural networks learn is by error attribution, which means that given a shared error, the network needs to figure out which weights contributed (so they can be adjusted) and which weights did NOT contribute (so they can be left alone). 1 0 1 0 1 1 0 0 1 1 1 1 0 1 1 1 0 1 0 1 0 1 1 0 Consider the first training example. Because the middle input is a 0, then the middle weight is completely irrelevant for this prediction. No matter what the weight is, it's going to be multiplied by zero (the input). Thus, any error at that training example (regardless of whether its too high or too low), can only be attributed to the far left and right weights. Consider the pressure of this first training example. If the network should predict 0, and two inputs are 1s, then this is going to cause error which drives the weight values towards 0. The "Weight Pressure" table helps describe the affect that each training example has on each respective weight. + indicates that it has pressure towards 1 whereas the - indicates that it has pressure towards 0. Zeroes (0) indicate that there is no pressure because the input datapoint is 0, so that weight won't be changed at all. Notice that the far left weight has 2 negatives and 1 positive, so on average the weight will move towards 0. The middle weight has 3 positives, so on average the weight will move towards 1. - 0 - 0 + + 0 0 - + + + 0 + + - 0 - 0 1 0 1 1 0 Training Data Weight Pressure Licensed to Asif Qamar 112 Chapter 6 I Building Your First "Deep" Neural Network Up and Down Pressure (cont.) 1 0 1 0 1 1 0 0 1 1 1 1 0 1 1 1 0 1 0 1 0 1 1 0 - 0 - 0 + + 0 0 - + + + 0 + + - 0 - 0 1 0 1 1 0 Training Data Weight Pressure So, each individual weight is attempting to compensate for error. In our first train ing example, we see discorrelation between the far right and left inputs and our desired output. This causes those weights to experience down pressure. This same phenomenon occurs throughout all 6 training examples, rewarding correlation with pressure towards 1 and penalizing de-correaltion with pressure towards 0. On average, this causes our network to find the correlation that is present between our middle weight and the output to be the dominant predictive force (heavest weight in the weighted average of our input) making our network quite accurate. Our prediction is a weighted sum of our inputs. Our learning algorithm rewards inputs that correlate with our output with upward pressure (towards 1) on their weight while rewarding inputs with no-correlation with downward pressure. So that our weighted sum of our inputs will find perfect correlation between our input and

our output, by weighting de-correlated inputs to 0. Bottom Line Now, the mathematician in you might be cringing a little bit. "upward pressure" and "down ward pressure" are hardly precise mathematical expressions, and they have plenty of edge cases where this logic doesn't hold (which we'll address in a second). However, we will later find that this is an extremely valuable approximation, allowing us to temporarily overlook all the complexity of Gradient Descent and just remember that learning rewards correlation with larger weights or more generally learning finds correlation between our two datasets. Licensed to Asif Qamar Edge Case: Overfitting 113 Edge Case: Overfitting Sometimes correlation happens accidentally... Error is shared between all of our weights. If a particular configuration of weights accidentally creates perfect correlation between our prediction and the output dataset (such that error == 0) without actually giving the heaviest weight to the best inputs... the neural network will stop learning. Deep Learning's Greatest Weakness: Overfitting Consider again the first example in the training data. What if our far left weight was 0.5 and our far right weight was -0.5. Their prediction would equal 0! The network would predict perfectly! However, it hasn't remotely learned how to safely predict streetlights (i.e. those weights would fail in the real world). This phenomenon is known as overfitting. In fact, if it wasn't for our other training examples, this fatal flaw would cripple our neural network. What do the other training examples do? Well, let's take a look at the second training example. It would bump the far right weight upward while not changing the far left weight. This throws off the equilibrium that stopped the learning in our first example. So, as long as we don't train exclusively on the first example, the rest of the training examples will help the network avoid getting stuck in these edge case configurations that exist for any one training example. This is super, super important. Neural networks are so flexible that they can find many, many different weight configurations that will correctly predict for a subset of your training data. In fact, if we trained our neural network on the first 2 training examples, it would likely stop learning at a point where it did NOT work well for our other training examples. In essence, it memorized the two training examples instead of actually finding the correlation that will generalize to any possible streetlight configuration. If we only train on two streetlights... and the network just finds these edge case configurations... it could FAIL to tell us whether it is safe to cross the street when it sees a streetlight that wasn't in our training data! The greatest challenge you will face with deep learning is convincing your neural network to generalize instead of just memorize. We will see this again. Licensed to Asif Qamar 114 Chapter 6 I Building Your First "Deep" Neural Network Edge Case: Conflicting Pressure Sometimes correlation fights itself. Consider the far right column in the "Weight Pressure" table below. What do you see? This column seems to have an equal number of upward and downward pressure moments. How ever, we have seen that the network correctly pushes this (far right) weight down to 0 which means that the downward pressure moments must be larger than the upward ones. How does this work? 1 0 1 0 1 1 0 0 1 1 1 1 0 1 1 1 0 1 0 1 0 1 1 0 - 0 - 0 + + 0 0 - + + + 0 + + - 0 - 0 1 0 1 1 0 Training Data Weight Pressure The left and middle weights have enough signal to converge on their own. The left weight falls to 0 and the middle weight moves towards 1. As the middle weight moves higher and higher, the error for positive examples continues to decrease. However, as they approach their optimal positions, the de-correlation on the far right weight becomes more apparent. Let's consider the extreme example of this, where the left and middle weights are perfectly set to 0 and 1 respectively. What happens to our network? Well, if our right weight is above 0, then our network predicts too high and if our right weight is beneath 0, our network pre dicts too low. In short, as other neurons learn, they absorb some of the error. They absorb some part of the correlation. They cause the network to predict with moderate correlative power which reduces the error. The other weights then only try to adjust their weights to correctly predict what's left! In this case, because the middle weight has consistent signal to absorb all of the correlation (because of the 1:1 relationship between the middle input and the output), the error when we want to predict 1 becomes very small but the

error to predict 0 becomes large... pushing our middle weight downward. Licensed to Asif Qamar Edge Case: Conflicting Pressure 115 Edge Case: Conflicting Pressure (cont.) It doesn't always work out like this. In some ways, we kindof got lucky. If our middle node hadn't been so perfectly correlated, our network might have struggled to silence our far right weight. In fact, later we will learn about Regularization which forces weights with conflicting pressure to move towards 0. As a preview, regularization is advantageous because if a weight has equal pressure upward and downward, then it isn't really good for anything. It's not helping either direc tion. In essence, regularization aims to say "only weights with really strong correlation can stay on... everything else should be silenced because its contributing noise". It's sortof like natural selection... and as a side effect it would cause our neural network to train faster (fewer iterations) because our far right weight has this "both positive and negative" pressure problem. In this case, because our far right node isn't definitively correlative, the network would immediately start driving it towards 0. Without regularization (like we trained it before), we won't end up learning that the far right input is useless until after the left and middle start to figure their patterns out. More on this later. So, if networks look for correlation between an input colum of data and our output column, what would our neural network do with this dataset? 1 0 1 0 1 1 0 0 1 1 1 1 1 1 0 0 + 0 + 0 + + 0 0 - - - - 1 1 0 0 Training Data Weight Pressure There is no correlation between any input column and the output column. Every weight has an equal amount of upward pressure as it does downward pressure. This data set is a real problem for our neural network. Previously, we could solve for input datapoints that had both upward and downward pressure because other neurons would start solving for either the positive or negative predictions... drawing our balanced neuron to favor up or down. However, in this case, all of the inputs are equally balanced between positive and negative pressure. What do we do? Licensed to Asif Qamar 116 Chapter 6 I Building Your First "Deep" Neural Network Learning Indirect Correlation If your data doesn't have correlation... let's create intermediate data that does! Previously, I have described neural networks as an instrument that searches for correlation between input and output datasets. I should refi ne this just a touch. In reality, neural networks actually search for correlation between their input and output layers. We set the values of our input layer to be individual rows of our input data... and we try to train the network so that our output layer equals our output dataset. Funny enough... the neural net work actually doesn't "know" about data. It just searches for correlation between the input and output layers. walk/stop .01 1.0 input -.0 output Unfortunately, we just encountered a new streetlights dataset where there isn't any correlation between our input and output. Th e solution is simple. Let's use two of these net works. Th e fi rst one will create an intermediate dataset that has limited correlation with our output. Th e second will then use that limited correlation to correctly predict our output! Since our input dataset doesn't correlate with our output dataset... we're going to use our input dataset to create an intermediate dataset that DOES have cor relation with our output. It's kindof like cheating! Licensed to Asif Qamar Creating Our Own Correlation 117 walk/stop layer_1 layer_2 layer_0 weights_0_1 weights_1_2 Creating Our Own Correlation If your data doesn't have correlation... let's create intermediate data that does! Below, you'll see a picture of our new neural network. Notice that we basically just stacked two neural networks on top of each other. Th e middle layer of nodes (layer_1) rep resents our intermediate dataset. Our goal is to train this network so that even though there's no correlation between our input dataset and output dataset (layer_0 and layer_2) that our layer_1 dataset that we create using layer_0 will have correlation with layer_2. Th ings to notice: Th is network is still just a function! It still just has a bunch of weights that are collected together in a particular way. Furthermore, Gradient Descent still works be cause we can calculate how much each weight contributes to the error and adjust it to reduce the error to 0. Th at's exactly what we're going to do. (this will be our "intermediate data") Licensed to Asif Qamar 118 Chapter 6 I Building Your First "Deep" Neural Network Stacking Neural Networks - A Review In Chapter 3, we briefl y

mentioned stacked neural networks. Let's review. So, when you look at the architecture below, the prediction occurs exactly as you might ex pect when I say "stack neural networks". Th e output of the fi rst "lower" network (layer_0 to layer_1) is the input to the second "upper" neural network (layer_1 to layer_2). Th e predic tion for each of these networks is identical to what we saw before. walk/stop layer_1 layer_2 layer_0 weights_0_1 weights_1_2 So, as we start to think about how this neural network learns, we actually already know a great deal. If we ignored the lower weights and just considered their output to be our training set, then the top half of the neural network (layer_1 to layer_2) is just like the networks we trained in the last chapter. We can use all the same learning logic to help them learn. In fact, this is the case. So, the part that we don't yet understand is how to update the weights between layer_0 and layer_1. What do they use as their error measure? If you remember from the last chapter, our cached/normalized error measure was called delta. In our case, we want to fi gure out how to know the delta values at layer_1 so that they can help layer_2 make accurate predictions. Licensed to Asif Qamar Backpropagation: Long Distance Error Attribution 119 layer_0 weights_0_1 Backpropagation: Long Distance Error Attribution The "weighted average error" What is the prediction from layer_1 to layer_2? It's just a weighted average of the values at layer_1. So, if layer_2 is too high by "x" amount. How do we know which values at layer_1 contributed to the error? Well, the ones with higher weights (weights_1_2) contribut ed more! Th e ones with lower weights from layer_1 to layer_2 contributed less! Consider the extreme. Let's say that the far left weight from layer_1 to layer_2 was zero. How much did that node at layer_1 cause the network's error? ZERO! It's so simple it's almost hilarous. Our weights from layer_1 to layer_2 exactly describe how much each layer_1 neuron contributes to the layer_2 prediction. Th is means that those weights ALSO exactly describe how much each layer_1 neuron contributes to the layer_2 error! So, how do we use the delta at layer_2 to fi gure out the delta at layer_1? We just multiply it by each of the respective weights for layer_1!!! It's like our prediction logic in reverse! Th is process of "moving delta signal around" is called backpropagation. +0.25 layer_1 layer_2 weights_1_2 this value is layer_2 delta (goal_prediction - prediction) 0.0 1.0 0.5 -1.0 0.0 0.125 0.25 -0.25 layer 1 deltas which are actually just "weighted" versions of our layer_2 delta. (I made up some weight values so you can see how the layer_2 delta passes through them) Licensed to Asif Qamar 120 Chapter 6 I Building Your First "Deep" Neural Network layer_0 weights_0_1 Backpropagation: Why does this work? The "weighted average delta" In our neural network from the previous chapter, our delta variable told us "the direction and amount we want the value of this neuron to change next time". All backprop agation lets us do is say "hey... if you want this neuron to be X amount higher... then each of these previous 4 neurons need to be X*weights_1_2 amount higher/lower... becuase these weights were amplifying the prediction by weights_1_2 times". When used in reverse, our weights_1_2 matrix amplifi es the error by the appropriate amount. It amplifi es the error so that we know how much each layer_1 node should move up or down. Once we know this, we can just update each weight matrix just like we did before. For each weight, multiply its output delta by its input value... and adjust our weight by that much. (or we can scale it with alpha). +0.25 layer_1 layer_2 weights_1_2 this value is layer_2 delta (goal_prediction - prediction) 0.0 1.0 0.5 -1.0 0.0 0.125 0.25 -0.25 layer 1 deltas which are actually just "weighted" versions of our layer_2 delta. (I made up some weight values so you can see how the layer_2 delta passes through them) Licensed to Asif Qamar Linear vs Non-Linear 121 Linear vs Non-Linear This is probably the hardest concept in the book. Let's take it slow. I'm going to show you a phenomenon. As it turns out, we need one more "piece" to make this neural network train. We're going to take it from two perspectives. Th e fi rst is gong to show you why the neural network can't train without it. In other words, fi rst I'm going to show you why our neural network is currently broken. Th en, once we add this piece, I'm going to show you what it does to fi x this problem. For now, check out this simple algebra. 1 * 10 * 2 = 100 5 * 20 = 100 Here's the takeaway, for any two multiplications that I do, I can actually accomplish the

same thing using a single multiplication. As it turns out, this is bad. Check out the following. 1.0 -1.0 0.25 0.25 -0.25 0.9 0.225 -0.225 1 * 0.25 * 0.9 = 0.225 1 * 0.225 = 0.225 1.0 -1.0 0.225 0.225 -0.225 Th ese two graphs show you two training examples each, one where the input is 1.0 and another where the input is -1.0. Here's the bot tom line, for any 3-layer network we create, there's a 2-layer network that has identical behavior. It turns out that just stacking two neural nets (as we know them at the moment) doesn't actually give us any more power! Two consecutive weighted sums is just a more ex pensive version of one weighted sum. Licensed to Asif Qamar 122 Chapter 6 I Building Your First "Deep" Neural Network Why The Neural Network Still Doesn't Work If we trained the 3 layer network as it is now, it would NOT converge. Problem: For any two consecutive weighted sums of the input, there exists a single weighted sum with exactly identical behavior. Aka... anything that our 3 layer network can do... our 2 layer network can also do. Let's talk about the middle layer (layer_1) that we have at present (before we fi x it). Right now, each node (out of the 4 we have), has a weight coming to it from each of the inputs. Let's think about this from a correlation standpoint. Each node in our middle layer subscribes to a certain amount of correlation with each input node. If the weight from an in put to the middle layer is a 1.0, then it subscribes to exactly 100% of that node's movement. If that node goes up by 0.3, our middle node will follow. If the weight connecting two nodes is 0.5, it subscribes to exactly 50% of that node's movement. Th e only way that our middle node can escape the correlation of one particular input node is if it subscribes to additional correlation from another input node. So, you can see, there's nothing new being contributed to this neural network. Each of our hidden nodes simply subscribe to a little bit of correlation from our input nodes. Our middle nodes don't actu ally get to add anything to the conversation. Th ey don't get to have correlation of their own. Th ey're just more or less correlated to various input nodes. However, since we KNOW that in our new dataset... there is NO correlation between ANY of our inputs and our output... then how could our middle layer help us at all?!?! It just gets to mix up a bunch of cor relation that was already useless! What we really need is for our middle layer to be able to selectively correlate with our input. We want it to sometimes correlate with an input, and sometimes not correlate. Th at gives it correlation of it's own! Th is gives our middle layer the opportunity to not just "always be X% correlated to one input and Y% correlated to another input". Instead, it can be "X% correlated to one input.... only when it wants to be... but other times not be correlated at all!". Th is is "conditional correlation" or "sometimes correlation". Licensed to Asif Qamar The Secret to "Sometimes Correlation" 123 The Secret to "Sometimes Correlation" We're going to simply turn our node "off" when the value would be below 0. This might seem too simple to work, but consider this. If the node's value dropped below 0, normally the node would still have just as much correlation to the input as it always did! It would just happen to be negative in value. However, if we turn off the node (setting it to 0) when it would be negative, then it has ZERO CORRELATION to ANY INPUTS when ever it's negative. What does this mean? It means that our node can now selectively pick and choose when it wants to be correlated to something. This allows it to say something like "make me perfectly correlated to the left input but ONLY when the right input is turned OFF". How would it do this? Well, if the weight from the left input is a 1.0, and the weight from the right input is a HUGE NEGATIVE NUMBER, then turning on both the left and right inputs would cause the node to just be 0 all the time. However, if just the left node was on, the node would take on the value of the left node. This wasn't possible before! Before our middle node was either ALWAYS correlat ed to an input or ALWAYS not correlated! Now it can be conditional! Now it can speak for itself! Solution: By turning any middle node off whenever it would be negative, we al low the network to sometimes subscribe to correlation from various inputs. This is impossible for 2-layer neural networks... thus adding power to 3-layer nets. The fancy term for this "if the node would be negative then set it to 0" logic is called a nonlinearity. This is because without this tweak, our neural network is linear. Without this technique, our output layer only gets to pick from the same

correlation that it had in the 2-layer network. It's still just subscribing to pieces of the input layer, which means that it can't solve our new streetlights dataset. There are many kinds of nonlinearities. However, the one we discussed above is, in many cases, the best one to use. It's also the simplest. (It's called "relu") For what it's worth, most other books/courses simply just say "consecutive matrix multiplication is still just a linear transformation". I find this to be very unintuitive. Further more, it makes it harder to understand what nonlinearities actually contribute and why you choose one over the other (which we'll get to later). It just says "without the nonlinearity two matrix multiplications might as well be 1". So, this page's explanation, while not the most terse answer, is an intuitive explanation of why we need nonlinearities. Licensed to Asif Qamar 124 Chapter 6 I Building Your First "Deep" Neural Network A Quick Break That last part probably felt a little abstract... that's totally ok. Let's chat for a sec. So, here's the deal. In previous chapters we were working with very simple algebra. This meant that everything was ultimately grounded in fundamentally simple tools. This chapter has started building on the presmises we learned previously. In other words, previously we learned lessons like: We can compute the relationship between our error and any one of our weights so that we know how changing the weight changes the error. We can then use this to reduce our error down to 0. That was a massive lesson! However, now we're moving past it. Since we already worked through why that works, we can just trust it. We take the statement at face value. The next big lesson came at the beginning of this chapter: Adjusting our weights to reduce our error over a series of training examples ultimately just searches for correlation between our input and our output layers. If no correlation exists, then error will never reach 0. This lesson is an even bigger lesson! Why? Well, it largely means that we can put the pre vious lesson out of our minds for now. We don't actually need it. Now we're focused on correlation. The takeaway for you is that you can't constantly think about everything all at once. You take each one of these lessons and you let yourself trust it. When it's a more con cise summarization... a higher abstraction... of more granular lessons, we can set aside the granular and only focus on understanding the higher summarizations. This is akin to a professional swimmer, biker, or really any other skill that requires a combined fluid knowledge of a bunch of really small lessons. A baseball player who swings a bat actually learned thousands of little lessons to ultimately culminate in a great bat swing. However, he doesn't think of all of them when he goes to the plate! He just lets it be fluid... subcontious even. It is the same way for studying these math concepts. Neural networks look for correlation between input and output... and you no longer have to worry about how that happens. We just know that it does. Now we're building on that idea! Let yourself relax and trust the things you've already learned. Licensed to Asif Qamar Our First "Deep" Neural Network 125 Our First "Deep" Neural Network How to Make the prediction In the code below, we initialize our weights and make a forward propagation. New is bold. import numpy as np np.random.seed(1) def relu(x): return (x > 0) * x alpha = 0.2 hidden_size = 4 streetlights = np.array([[1, 0, 1], [0, 1, 1], [0, 0, 1], [1, 1, 1]]) walk_vs_stop = np.array([[1, 1, 0, 0]]).T weights_0_1 = 2*np.random.random((3,hidden_size)) - 1 weights_1_2 = 2*np.random.random((hidden_size,1)) - 1 layer_0 = streetlights[0] layer_1 = relu(np.dot(layer_0,weights_0_1)) layer_2 = np.dot(layer_1,weights_1_2) 2 sets of weights now to connect our 3 layers (randomly initialized) the output of layer_1 is sent through "relu" where negative values become 0. This is then the input for the next layer, layer_2 layer_1 layer_2 layer_0 weights_0_1 weights_1_2 Take each piece and follow along with the picture on the bottom right. Input data comes into lay er_0. Via the "dot" function, the signal travels up the weights from layer_0 to layer_1 (performing a weighted sum at each of the 4 lay er_1 nodes). Th ese weighted sums at layer_1 are then passed through the "relu" function, which converts all negative numbers to zero. We then perform a fi nal weighted sum into the fi nal node, layer_2. this function sets all negative numbers to 0 Licensed to Asif Qamar 126 Chapter 6 I Building Your First "Deep" Neural Network Backpropagation in Code How we can learn the amount that each weight contributes to the

final error. At the end of the previous chapter, I made an assertion that it would be very important to memorize the 2-layer neural network code so that you could quickly and easily recall it when I reference the more advanced concepts. This is when that memorization maters! We're about to look at the new learning code and it is absolutely essential that you recognize and understand the parts that were addressed in the previous chapters. If you get lost, go back to the last chapter and memorize the code and come back. It'll save your life someday. import numpy as np np.random.seed(1) def relu(x): return (x > 0) * x # returns x if x > 0 # return 0 otherwise def relu2deriv(output): return output>0 # returns 1 for input > 0 # return 0 otherwise alpha = 0.2 hidden_size = 4 weights_0_1 = 2*np.random.random((3,hidden_size)) - 1 weights_1_2 = 2*np.random.random((hidden_size,1)) - 1 for iteration in xrange(60): layer_2_error = 0 for i in xrange(len(streetlights)): layer_0 = streetlights[i:i+1] layer_1 = relu(np.dot(layer_0,weights_0_1)) layer_2 = np.dot(layer_1,weights_1_2) layer_2_error += np.sum((layer_2 - walk_vs_stop[i:i+1]) ** 2) layer_2_delta = (walk_vs_stop[i:i+1] - layer_2) layer_1_delta=layer_2_delta.dot(weights_1_2.T)*relu2deriv(layer_1) weights_1_2 += alpha * layer_1.T.dot(layer_2_delta) weights_0_1 += alpha * layer_0.T.dot(layer_1_delta) if(iteration % 10 == 9): print "Error:" + str(layer_2_error) Believe it or not, the only truly new code is in bold. Everything else is fundamentally the same as in previous pages. The "relu2deriv" function returns 1 when "output" is > 0 and it returns 0 otherwise. This is actually the slope of our relu function. It's the derivative of our relu function. It serves a very important purpose as we'll see in a moment. this line computes the delta at layer_1 given the delta at layer_2 by taking the layer_2_delta and multiplying it by its connecting weights_1_2 Licensed to Asif Qamar Backpropagation in Code 127 layer_0 weights_0_1 +0.25 layer_1 layer_2 weights_1_2 this value is layer_2 delta (goal_prediction - prediction) 0.0 1.0 0.5 -1.0 0.0 layer 1 deltas which are actually just "weighted" versions of our layer_2 delta. (I made up some weight values so you can see how the layer_2 delta passes through them) Remember, the goal here is error attribution. It's all about fi guring out how much each weight contributed to the fi nal error. In our fi rst (2-layer) neural network, we calculat ed a delta variable, which told us how much higher or lower we wanted the output predic tion to be. Look at the code here. We compute our layer_2_delta in the same way. Nothing new here! (again, go back to the previous chapter if you've forgotten how that part works) So, now that we have how much we want the fi nal prediction to move up or down (delta), we need to fi gure out how much we want each middle (layer_1) node to move up or down. Th ese are eff ectively intermediate predictions. Once we have the delta at layer_1, we can use all the same processes we used before for calculating a weight update (for each weight, multiply its input value by its output delta and increase the weight value by that much). So, how do we calculate the deltas for layer_1? Well, fi rst we do the obvious as mentioned on the previous pages, we multiply the output delta by each weight attached to it. Th is gives us a weighting of how much each weight contributed to that error. Th ere's one more thing we need to factor in. If the relu set the output to a layer_1 node to be 0, then it didn't contribute to the error at all. So, when this was true, we should also set the delta of that node to be zero. Multiplying each layer_1 node by the relu2deriv function accomplishes this. relu2deriv is either a 1 or a 0 depending on whether the layer_1 value was > 0 or not. 0.125 0.25 -0.25 Licensed to Asif Qamar 128 Chapter 6 I Building Your First "Deep" Neural Network 1 Initialize the Network's Weights and Data inputs prediction hiddens import numpy as np np.random.seed(1) def relu(x): return (x > 0) * x def relu2deriv(output): return output>0 lights = np.array([[1, 0, 1], [0, 1, 1], [0, 0, 1], [1, 1, 1]]) walk_stop = np.array([[1, 1, 0, 0]]).T alpha = 0.2 hidden_size = 3 weights_0_1 = 2*np.random.random(\ (3,hidden_size)) - 1 weights_1_2 = 2*np.random.random(\ (hidden_size,1)) - 1 One Iteration of Backpropagation 2 PREDICT & COMPARE: Make a Prediction, Calculate Output Error and Delta inputs prediction hiddens layer_0 = lights[0:1] layer_1 = np.dot(layer_0,weights_0_1) layer_1 = relu(layer_1) layer_2 = np.dot(layer_1,weights_1_2) error = (layer_2-walk_stop[0:1])**2

layer_2_delta=(layer_2-walk_stop[0:1]) 1 0 1 0 0 .13 -.02 1.04 0.14 layer_0 layer_2 layer_1 Licensed to Asif Qamar One Iteration of Backpropagation 129 3 LEARN: Backpropagate From layer_2 to layer_1 inputs prediction hiddens layer_0 = lights[0:1] layer_1 = np.dot(layer_0,weights_0_1) layer_1 = relu(layer_1) layer_2 = np.dot(layer_1,weights_1_2) error = (layer_2-walk_stop[0:1])**2 layer_2_delta=(layer_2-walk_stop[0:1]) 1 0 1 0 0 .13 -.02 1.04 0.14 layer_0 layer_2 layer_1 -.17 0 0 layer_1_delta=layer_2_delta.dot(weights_1_2.T) layer_1_delta *= relu2deriv(layer_1) 4 LEARN: Generate Weight Deltas and Update Weights inputs prediction hiddens layer_0 = lights[0:1] layer_1 = np.dot(layer_0,weights_0_1) layer_1 = relu(layer_1) layer_2 = np.dot(layer_1,weights_1_2) error = (layer_2-walk_stop[0:1])**2 layer_2_delta=(layer_2-walk_stop[0:1]) 1 0 1 0 0 .13 -.02 1.04 0.14 layer_0 layer_1 layer_2 -.17 0 0 layer_1_delta=layer_2_delta.dot(weights_1_2.T) layer_1_delta *= relu2deriv(layer_1) weight_delta_1_2 = layer_1.T.dot(layer_2_delta) weight_delta_0_1 = layer_0.T.dot(layer_1_delta) weights_1_2 -= alpha * weight_delta_1_2 weights_0_1 -= alpha * weight_delta_0_1 As we can see, backpropagation in its entirety is about calculating deltas for inter mediate layers so that we can perform Gradient Descent. In order to do so, we simply take the weighted average delta on layer_2 for layer_1 (weighted by the weights inbetween them). We then turn off (set to 0) nodes that weren't participating in the forward prediction, since they could not have contributed to the error. Licensed to Asif Qamar 130 Chapter 6 I Building Your First "Deep" Neural Network Putting it all together Here's the self sufficient program you should be able to run (runtime output below) import numpy as np np.random.seed(1) def relu(x): return (x > 0) * x # returns x if x > 0 # return 0 otherwise def relu2deriv(output): return output>0 # returns 1 for input > 0 # return 0 otherwise streetlights = np.array([[1, 0, 1], [0, 1, 1], [0, 0, 1], [1, 1, 1]]) walk_vs_stop = np.array([[1, 1, 0, 0]]).T alpha = 0.2 hidden_size = 4 weights_0_1 = 2*np.random.random((3,hidden_size)) - 1 weights_1_2 = 2*np.random.random((hidden_size,1)) - 1 for iteration in xrange(60): layer_2_error = 0 for i in xrange(len(streetlights)): layer_0 = streetlights[i:i+1] layer_1 = relu(np.dot(layer_0,weights_0_1)) layer_2 = np.dot(layer_1,weights_1_2) layer_2_error += np.sum((layer_2 - walk_vs_stop[i:i+1]) ** 2) layer_2_delta = (layer_2 - walk_vs_stop[i:i+1]) layer_1_delta=layer_2_delta.dot(weights_1_2.T)*relu2deriv(layer_1) weights_1_2 -= alpha * layer_1.T.dot(layer_2_delta) weights_0_1 -= alpha * layer_0.T.dot(layer_1_delta) if(iteration % 10 == 9): print "Error:" + str(layer_2_error) Error:0.634231159844 Error:0.358384076763 Error:0.0830183113303 Error:0.0064670549571 Error:0.000329266900075 Error:1.50556226651e-05 Licensed to Asif Qamar Why do deep networks matter? 131 Why do deep networks matter? What's the point of creating "intermediate datasets" that have correlation? Consider the cat picture below. Consider further that we had a dataset of images "with cats" and "without cats" (and we labeled them as such). If we wanted to train a neural network to take our pixel values and predict whether or not there is a cat in the picture, our 2-layer network might have a problem. Just like in our last streetlight dataset, no individual pixel correlates with whether or not there is a cat in the picture. Only diff erent confi gurations of pixels correlate with whether or not there is a cat. Th is is the essence of Deep Learning. Deep Learning is all about creating interme diate layers (datasets) wherein each node in an intermediate layer represents the presence or absence of a diff erent confi guration of inputs. In this way for our cat images dataset, no individual pixel has to correlate with whether or not there is a cat in the photo. Intead, our middle layer would attempt to identify diff erent confi gurations of pixels that may or may not correlate with a cat (such as an ear, or cat eyes, or cat hair). Th e presence of many "cat like" confi gurations would then give the fi nal layer the information (correlation) it needs to correctly predict the presence of absence of a cat! Believe it or not, we can take our 3-layer network and continue to stack more and more layers. Some neural networks even have hundreds of layers, each neuron playing its part in detecting diff erent confi gurations of input data. Th e rest of this book will be dedicat ed to studying diff erent phenomena within these layers in an eff ort to explore the full

power of deep neural networks. To that end, I must issue the same challenge as I did in the previous chapter. Mem roze the code on the previous page. You will need to be very familiar with each of the opera tions in the code in order for the following chapters to be readable. Do not progress past this page until you can build a 3 layer neural network from memory!!!